COMP6202 - Evolution of Complexity Assignment 2

Supritha Konaje - 32864477
ssk1n21@soton.ac.uk

Abstract

This paper talks about reproducing the experiments discussed in *"Global Adaptation in Networks of Selfish Components: Emergent Associative Memory at the System Scale"*. The first few sections of the paper discusses the re-implementation part of the paper. The last part talks about the extension that was done related to this implementation.

1 Introduction

Networking is initialised when the agents in an environment start interacting with each other. If an agent finds a particular agent in that network to be useful for their adaptivity, they give them importance and start discarding the ones that are not of much importance. This agent can be in the form of a person, trait or the chromosomes. Be it any environment, agents are not aware on how to adapt. An agent's strategy mainly depends on the environment around it. Basically, if the other agent has comparatively good utility, then it tries to achieve a better utility by learning selfishly by modifying the network connections [1].

2 Reproducibility

We will design a model that will selfishly increase it's utility by obeying Hebb's rule. We are considering a distributed environment, where multiple agents will interact and try to selfishly improve their utility which will also directly impact the environment connections.

We consider two agents that is s_i and s_j who interact with each other and

their interaction or connection strength is maintained in ω_{ij}. If there is connectivity between the two agent then the value of $\omega_{ij} = 1$ else it will be 0[1]. The fitness of the the connection [1] of an agent can be determined by,

$$u_i = \sum_{n=j}^{N} \omega_{ij} s_i s_j$$

Global payoff of the system [1] is given by,

$$U = \sum_{n=i}^{N} \sum_{n=j}^{N} \omega_{ij}^{o} s_i s_j$$

To find which utility is better, we subtract the new utility with the old utility and if the new utility is better then it will be kept and the old one will be discarded. This way the agent makes it's connection stronger with the new agent and weakens its connection with the old agent. To find the agent's utility after Hebb's rule, we need to consider the learning rate δ. Using the general form of Hebb's rule $\Delta\omega_{ij} = \delta s_i s_j$ [2], if $\delta > 0$ then there is Hebbian learning else it is not a Hebbian Learning.

We use the Simulated Annealing End of Relaxation algorithm for learning where we select the fittest among the agent's utility using the heuristic proposed [1]. If we look at the flow of selecting the fittest for adapting to the environment, it can be said that we will be using the Entropy-Boltzmann Selection in which the fitness does not stay fixed and keeps varying according to the environment.

Table 1: Below given table shows the parameter used to reproduce the graph

Parameter	S1	S2
n (shape of matrix)	(30,30)	(10,10)
Relaxation Period (N)	500	150
Learning Rate	0.0002	0.002
Relaxation Rate	e(N)	e(N)
k	8	5
p	-	0.01

3 Re-implementation Results

In this section, Figure 1 and Figure 2 show the comparison of results that were plotted for utility against their frequency for original as well as after selfish modification to the connections.

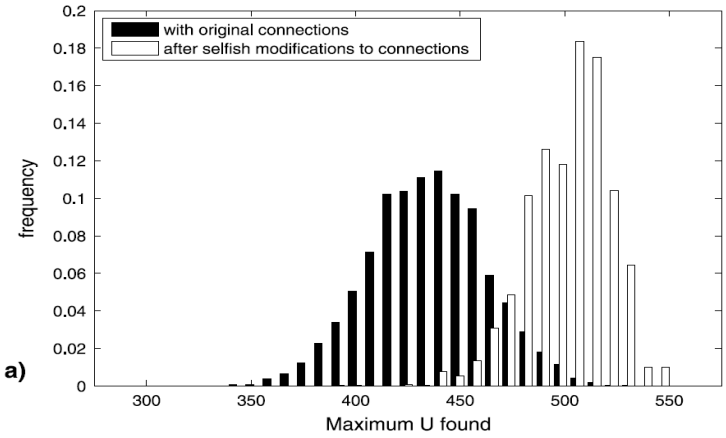

Figure 1: Histogram from the paper for S1

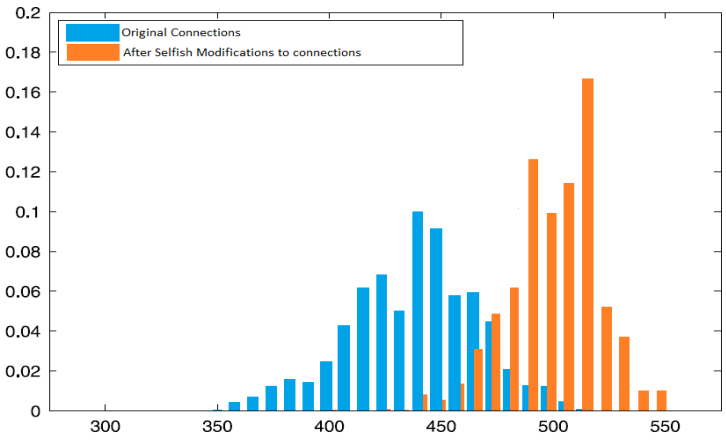

Figure 2: Re-implemented results for S1

In Figure 1, initially S1 is random sparse symmetric matrix. Over the period of relaxations updates are done for this matrix and then again their

utility is calculated. Both the graphs look similar and the results obtained are satisfactory.

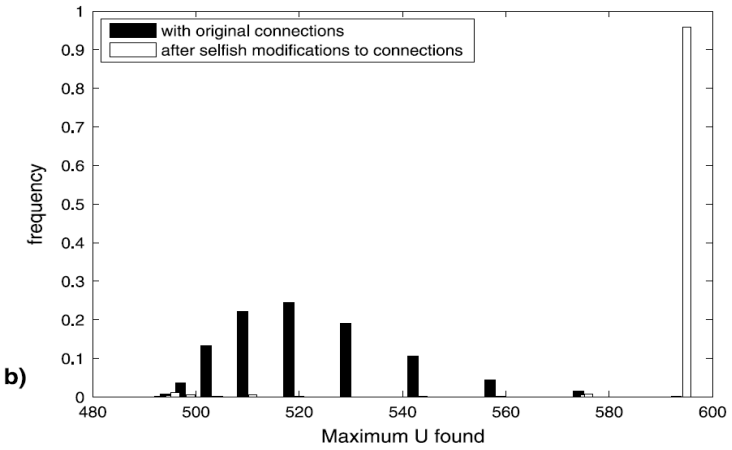

Figure 3: Histogram from the paper for S2

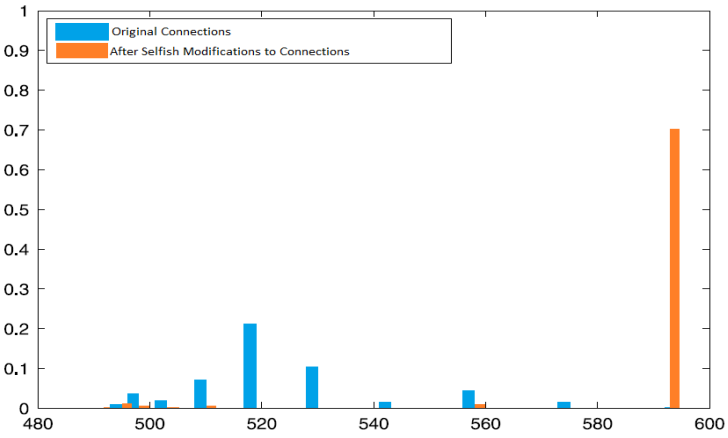

Figure 4: Re-implemented results for S2

In Figure 2, initially S2 is a modular structured matrix. The global payoff utility after selfish modifications is comparatively better for this structure.

4 Extension

In this section we make an extension to the author's research. In the related work section, the author has mentioned that we can explore the conditions where the initial matrix that we define as sparse symmetric or modular can be made semi-modular.

When in an environment, agents can adapt any quality which has more global payoff. Now an agent can be made a mix of many features. One of the feature that we can consider is semi-modularity.

We have defined a semi-modular structure as a mixture of both the extremes in which the matrix is symmetric and with a probability of 0.5, it has both the positive and negative number. This way the system has become semi-modular and has no initial zeros. The number of elements in a matrix (N) was 100, learning rate was 0.002 and it makes changes after 150 relaxations.

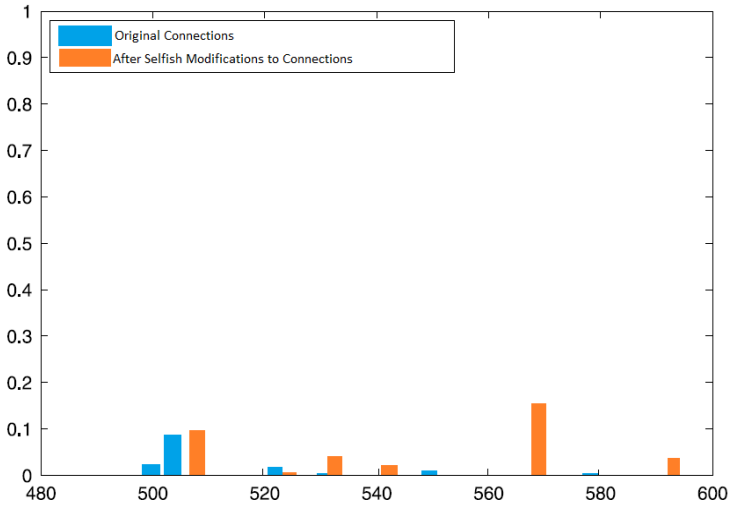

Figure 5: S3 with initially a Semi-Modular Structure

In Figure 3, it is seen that the utility is sparse and there is no consistent utility for a considerable frequency.

5 Conclusion

To conclude, the extension shows that not all type of agent structure can perform selfishly in the given algorithm. For future work, we can try by changing the parameters that are declared as constants across the system. It can also be tried on a system which is not strongly correlated.

References

[1] R. Watson, R. Mills, and C. Buckley, "Global adaptation in networks of selfish components: Emergent associative memory at the system scale," *Artificial life*, vol. 17, pp. 147–66, 05 2011.

[2] G. L. Shaw, "Donald hebb: The organization of behavior," in *Brain Theory*, G. Palm and A. Aertsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 231–233.

A Appendix

A.1 utils.py

```
import math
import numpy as np
import scipy.sparse as sparse
from numpy.random.mtrand import RandomState


def sprandsym(n=30, density=0.06):
    s1 = sparse.random(n, n, density=density, random_state=
    RandomState(1))
    return s1.toarray()


def modconmatrix(n, k=5, p=0.01):
    s2 = np.random.rand(n, n)
    return s2


def semimodmatrix(n):
```

```
    s2 = np.random.rand(n, n)
    for i in range(s2.shape[0]):
        for j in range(s2.shape[1]):
            if np.random.uniform(0, 1) < 0.5:
                s2[i][j] = -s2[i][j]
    return s2


def initialiseweightsforsymmetricmatrix(n=30):
    weightmatrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if np.random.uniform(0, 1) < 0.5:
                weightmatrix[i][j] = -1
            else:
                weightmatrix[i][j] = 1
    return weightmatrix


def randomize_states(n):
    np.random.seed()
    s = np.array((n, n))
    for i in range(n):
        for j in range(n):
            if np.random.uniform(0, 1) < 0.5:
                s[i][j] = -1
            else:
                s[i][j] = 1


def initialiseweightsformodularconnmatrix(n=10, k=5, p=0.01):
    weightmatrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if (i / k) == (j / k):
                weightmatrix[i][j] = 1
            else:
                weightmatrix[i][j] = p

    return weightmatrix
```

```
def calculateutility(si, sj, w, n):
    utility = 0
    for a in range(n):
        for b in range(n):
            utility += np.abs(w[a][b] * si[a][b] * sj[a][b])
            # print(original_utility)
    return utility


def relaxation(N, n, si, sj, relax_rate, w):
    seq = [-1, 1]
    actual_state = np.zeros((n, n))

    relaxation_factor = math.exp(N) * (3 / 4) * math.log(N / 2)
    fitness = 0
    conn_with_change = 0
    conn_without_change = 0
    cumm_eff = 0

    old_actual_state = actual_state
    cumm_eff += (relax_rate) * np.linalg.norm(
    np.matmul(np.matmul(si, sj), actual_state) - old_actual_state)
    while relaxation_factor > cumm_eff:
        for i in range(w.shape[0]):
            for j in range(w.shape[1]):
                fitness = fitness + w[i][j]
                # print(relaxation_factor, cumm_eff)
                conn_without_change = conn_without_change
                + fitness * actual_state[i][j]
                * si[i][j] * sj[i][j]

                actual_state[i][j] = np.random.uniform(0, 1)
                conn_with_change = conn_with_change
                + fitness * (-actual_state[i][j])
                * si[i][j] * sj[i][j]

                if conn_without_change < conn_with_change:
                    old_actual_state = actual_state[i][j]
```

```
                    actual_state[i][j] = -actual_state[i][j]
                    cumm_eff += (relax_rate) * np.linalg.norm(
                        np.matmul(

                        np.matmul(si, sj), actual_state)
                        - old_actual_state)
    return actual_state


def hebbian_weight(weight):
    if weight < -1:
        return -1
    elif weight > 1:
        return 1
    else:
        return weight


def hebbian_learning(si, sj, w, lr):
    for p in range(w.shape[0]):
        for q in range(w.shape[1]):
            if w[p][q] is not None:
                w[p][q] = hebbian_weight(
                w[p][q] + (lr * si[p][q] * sj[p][q]))
    return w
```

A.2 S1.py

```
import numpy as np

import math

from utils import sprandsym,
initialiseweightsforsymmetricmatrix, hebbian_learning,
calculateutility, relaxation

if __name__ == '__main__':
    # For S1
```

```
si = sprandsym(30, 0.06)
sj = sprandsym(30, 0.06)
w = initialiseweightsforsymmetricmatrix(30)

lr_rate = 0.0002
N = 500
relax_rate = math.exp(N)

u_o = {}
u_o_before_change = {}
count_o = 0

u_l = {}
u_l_after_change = {}
count_l = 0

for i in range(N):
    count_o += 1
    utility = calculateutility(si, sj, w, 30)
    u_o[count_o] = utility
    w = relaxation(N, 30, si, sj, relax_rate, w)
    if count_o > 100:
        u_o_before_change[count_o] = np.mean(np.abs(u_o))
        u_o = {}
        count_o = 0

for i in range(N):
    count_l += 1
    w = hebbian_learning(si, sj, w, lr_rate)
    utility = calculateutility(si, sj, w, 30)
    u_l[count_l] = utility
    w = relaxation(N, 30, si, sj, relax_rate, w)
    if count_l > 100:
        u_l_after_change[count_l] = np.mean(np.abs(u_l))
        u_l = {}
        count_l = 0
```

A.3 S2.py

```
si = modconmatrix(10, 5, 0.01)
    sj = modconmatrix(10, 5, 0.01)
    w = initialiseweightsformodularconnmatrix(10)

    lr_rate = 0.002
    N = 150
    relax_rate = math.exp(150)

    u_o = {}
    u_o_before_change = {}
    count_o = 0

    u_l = {}
    u_l_after_change = {}
    count_l = 0

    for i in range(N):
        count_o += 1
        utility = calculateutility(si, sj, w, 10)
        u_o[count_o] = utility
        w = relaxation(N, 10, si, sj, relax_rate, w)
        if count_o > 100:
            u_o_before_change[count_o] = np.mean(np.abs(u_o))
            u_o = {}
            count_o = 0

    for i in range(N):
        count_l += 1
        w = hebbian_learning(si, sj, w, lr_rate)
        utility = calculateutility(si, sj, w, 10)
        u_l[count_l] = utility
        w = relaxation(N, 10, si, sj, relax_rate, w)
        if count_l > 100:
            u_l_after_change[count_l] = np.mean(np.abs(u_l))
            u_l = {}
            count_l = 0
```

A.4 S3.py

```
import numpy as np

import math

from utils import hebbian_learning,
calculateutility, relaxation, modconmatrix,
initialiseweightsformodularconnmatrix,
    semimodmatrix

if __name__ == '__main__':
    # For S2

    si = semimodmatrix(10)
    sj = semimodmatrix(10)
    w = initialiseweightsformodularconnmatrix(10)

    lr_rate = 0.002
    N = 150
    relax_rate = math.exp(150)

    u_o = {}
    u_o_before_change = {}
    count_o = 0

    u_l = {}
    u_l_after_change = {}
    count_l = 0

    for i in range(N):
        count_o += 1
        utility = calculateutility(si, sj, w, 10)
        u_o[count_o] = utility
        w = relaxation(N, 10, si, sj, relax_rate, w)
        if count_o > 100:
            u_o_before_change[count_o] = np.mean(np.abs(u_o))
            u_o = {}
            count_o = 0

    for i in range(N):
        count_l += 1
```

```
w = hebbian_learning(si, sj, w, lr_rate)
utility = calculateutility(si, sj, w, 10)
u_l[count_l] = utility
w = relaxation(N, 10, si, sj, relax_rate, w)
if count_l > 100:
    u_l_after_change[count_l] = np.mean(np.abs(u_l))
    u_l = {}
    count_l = 0
```